Introduction to Arrays

Contents

[I. Array 2](#_Toc199707374)

[1. Introduction 2](#_Toc199707375)

[What does "Linear" mean? 2](#_Toc199707376)

[2. Problems 3](#_Toc199707377)

[Question 1: Count Elements With At Least One Greater 3](#_Toc199707378)

[Question 2: Find a Pair with Given Sum 4](#_Toc199707379)

[Final Thought: Half of n² Is Still n² 7](#_Toc199707380)

[Question 3: Array Reversal In-Place 7](#_Toc199707381)

[Question 4: Reversing a Subarray 8](#_Toc199707382)

[Question 5: Rotating an Array K Times 9](#_Toc199707383)

[Reverse-Based Rotation 10](#_Toc199707384)

# Array

## Introduction

* An array is a **linear data structure** that stores a collection of **elements of the same data type**.
* The elements in an array are **stored contiguously in memory**.
* Key Properties:
  + **Homogeneous**: All elements have the same data type.
  + **Contiguous Storage**: Stored back-to-back in memory.
  + **Indexing**: Elements are accessed using indexes starting from 0.

### What does "Linear" mean?

* **Logically Linear**: Elements follow a sequence — one comes after the other.
* It's **not about memory layout**, but how we access the data.
* Arrays are linear, but trees and graphs are non-linear data structures because elements don't follow a strict sequence.

## Problems

### Question 1: Count Elements With At Least One Greater

* Given an array and you want to count the number of elements having at least one element greater than themselves.

Input: [-3, -2, 6, 8, 4, 8, 5]

* + Max element = 8
  + Elements not having any greater than themselves = All elements equal to 8
  + All others have at least one greater value (8)
* Final Answer = Total elements (7) − Count of max elements (2) = 5

#### Step-by-Step Strategy:

* Find the maximum value in the array.
* Count how many times the maximum value occurs.
* Compute answer as: answer = n - count\_of\_max

#### Implementation in Two Loops

1. Step 1: Find the maximum element
2. Step 2: Count frequency of the max element
3. Step 3: Return the answer as ‘n – count’

size\_t Approach1(const vector<int>& vec)

{

int maxElement = INT\_MIN;

for (size\_t i = 0; i < vec.size(); i++) {

if (vec[i] > maxElement)

maxElement = vec[i];

}

size\_t count = 0;

for (size\_t i = 0; i < vec.size(); i++) {

if (vec[i] == maxElement)

count++;

}

return vec.size() - count;

}

#### Single-Pass Implementation

* In a single loop, do both:

1. Track the max element seen so far
2. Maintain its count

size\_t Approach2(const vector<int>& vec) {

size\_t count = 0;

int maxElement = INT\_MIN;

for (size\_t i = 0; i < vec.size(); i++) {

if (maxElement < vec[i]) {

maxElement = vec[i];

count = 1;

}

else if (maxElement == vec[i]) {

count++;

}

}

return vec.size() - count;

}

### Question 2: Find a Pair with Given Sum

#### Problem Statement

* You're given an array of size n and an integer k.
* You must check (we have to return either true or false) whether there exists a pair of indices (i, j) such that:

arr[i] + arr[j] == k and i ≠ j

* Return true if such a pair exists, otherwise false.
* Example 1: arr = [3, -2, 1, 4, 3, 6, 8] and k = 10
  + Do any two different elements sum to 10?
    - Yes: 4 + 6 = 10
    - So, return: true
* Example 2: arr = [3, -2, 1, 4, 3, 6, 8] and k = 8
  + - Since 4 + 4 = 8, and i == j hence it is invalid.
    - So, return: false

#### Brute Force Approach (O(n²))

1. Loop over every possible pair of indices (i, j):
2. Check if arr[i] + arr[j] == k
3. Also ensure i ≠ j
4. If such a pair is found: return true
5. If no pair found after checking all: return false

bool hasPairWithSum1(const vector<int>& vec, int k)

{

const size\_t n = vec.size();

for (size\_t i = 0; i < n; i++) {

for (size\_t j = 0; j < n; j++) {

if (i != j) {

if (vec[i] + vec[j] == k) {

return true;

}

}

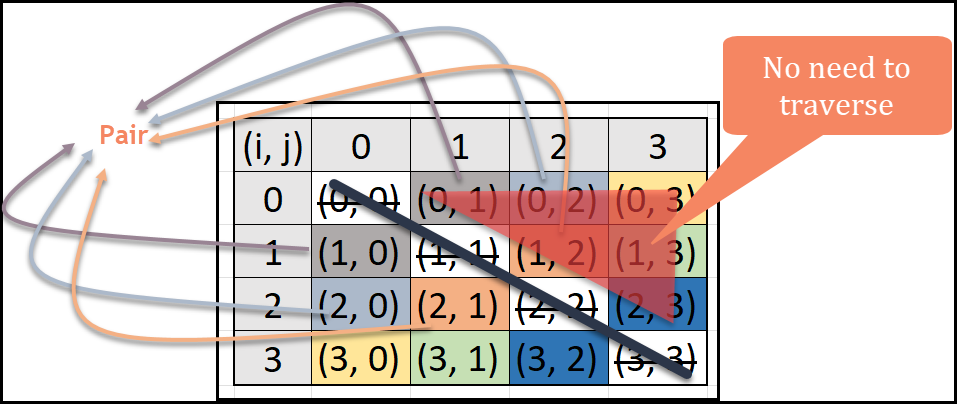
}

}

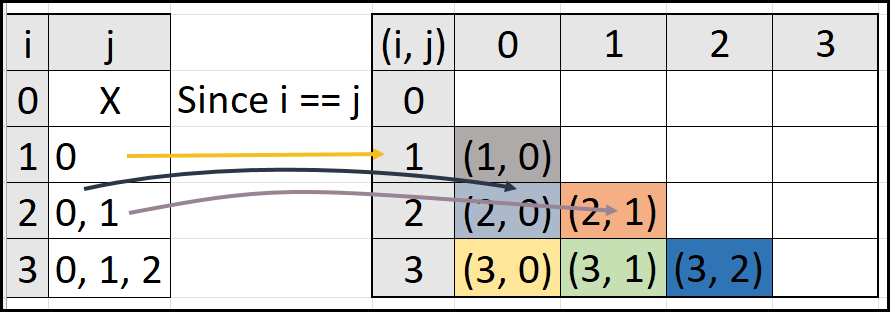
return false;

}

TC = outer loop \* inner loop

Problem: This checks both (i, j) and (j, i) — duplicates work. 

#### Optimized Version – Iterate Half Triangle

* We can iterate only half of the triangle as shown above.
* For n = 4, valid pairs are:

(1, 0)

(2, 0), (2, 1)

(3, 0), (3, 1), (3, 2)

* You avoid:
  + (0,0), (1,1), ...
  + Duplicates like (0,1) if (1,0) already checked

bool hasPairWithSum2(const vector<int>& vec, int k) {

const size\_t n = vec.size();

for (size\_t i = 1; i < n; i++) {

for (size\_t j = 0; j < i; j++) {

if (k == vec[i] + vec[j]) {

return true;

}

}

}

return false;

}

#### Time Complexity:

* Outer loop: runs (n - 1) times
* Inner loop: runs i times for each i

|  |  |  |
| --- | --- | --- |
| i (1 to n-1) | j (0 to i) | Total iterations |
| 1 | 0 | 1 |
| 2 | 0, 1 | 2 |
| 3 | 0, 1, 2 | 3 |
| 4 | 0, 1, 2, 3 | 4 |

* Neglecting lower order term
* Neglecting constant co-efficient

### Final Thought: Half of n² Is Still n²

* Even though you're doing **half the work**, when is large (e.g., ), half of n is still huge.
* Leads to **TLE (Time Limit Exceeded)** in coding contests.

### Question 3: Array Reversal In-Place

#### Problem Statement

* You are given an array, and you need to reverse it in-place, meaning:
  + Do not use an extra array. Reverse the array within the same memory.
* What Does "In-Place" Mean?
  + Modify the input directly, without using any additional data structures.
  + You're allowed to:
    - Use a few variables like i, j, temp.
  + You're not allowed to:
    - Create a new array and copy elements.

#### Step-by-Step Strategy:

* We use 2 pointers.
  + i points to the first element and j points to the last element in the array.
  + Swap the elements pointed by i and j.
  + Forward move i to point to next element and move backward j to point to second last element in the array.
* Two Pointer Technique

void reverseArray(vector<int>& vec)

{

size\_t i = 0;

size\_t j = vec.size() - 1;

while (i < j) {

std::swap(vec[i], vec[j]);

i++;

j--;

}

}

* + Set i = 0 (start of array)
  + Set j = n - 1 (end of array)
  + While i < j:
    - Swap arr[i] and arr[j]
    - Increment i, decrement j

#### Time and Space Complexity

### Question 4: Reversing a Subarray

* You are given:
  + An array.
  + A start index and an end index.
* You need to reverse the portion of the array between start and end in-place.
* Example:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Start: 2, End: 6 | | | | | | | |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Array | 1 | 2 | 3 | 4 | 3 | 1 | 2 |
| Result | 1 | 2 | 2 | 1 | 3 | 4 | 3 |

* Solution: Use the two-pointer approach:
  + Initialize i = start, j = end.
  + While i < j,
    - swap arr[i] and arr[j],
    - then increment i and decrement j.

void reverseSubArray(vector<int>& vec, size\_t start, size\_t end) {

while (start < end) {

std::swap(vec[start], vec[end]);

start++;

end--;

}

}

#### Time and Space Complexity

### Question 5: Rotating an Array K Times

* You are given an array and a number k. Rotate the array k times so that the **last elements move to the front**.
* Example:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Array | 3 | -2 | 1 | 4 | 6 | 9 | 8 |
| k = 1 | 8 | 3 | -2 | 1 | 4 | 6 | 9 |
| k = 2 | 9 | 8 | 3 | -2 | 1 | 4 | 6 |

* Constraints

#### Brute Force Idea:

* Write a function to rotate array by 1.
* Call that function K times.
* Correct, but Inefficient

#### Optimized Observations

* After K rotations:
  + The last K elements come to the front.
  + The first N-K elements shift to the back.

#### Optimized Solution (Using Extra Array)

* Create a new array.
* Copy last K elements to the front.
* Copy first N-K elements after them.

vector<int> rotateK\_TimesExtraSpace(vector<int>& vec, size\_t k) {

size\_t n = vec.size();

vector<int> newVec(n);

for (size\_t i = 0; i < k; i++) {

newVec[i] = vec[n - k + i];

}

for (size\_t i = k; i < n; i++) {

newVec[i] = vec[i - k];

}

return newVec;

}

##### Time and Space Complexity

### Reverse-Based Rotation

* Rotation can be reduced to 3 reversals. To rotate an array by k:

1. Reverse the entire array.
2. Reverse the first k elements.
3. Reverse the remaining n-k elements.

void rotateK\_Times(vector<int>& vec, size\_t k) {

if (k == 0)

return;

size\_t n = vec.size();

if (k >= n)

k = k % n;

reverseSubArray(vec, 0, n - 1);

reverseSubArray(vec, 0, k - 1);

reverseSubArray(vec, k, n - 1);

}

#### Time and Space Complexity